Design and implementation of a resource-secure system

Matthieu Lemerre
CEA LIST

Vincent David
CEA LIST

Guy Vidal-Naquet
SUPELEC

Abstract

This paper describes an operating system for safe execution of hard real-time and non real-time tasks on a single computer. Achieving this goal requires not only to follow the traditional *behavioral* security principles, but also new *resource* security principles throughout the system. Even if these principles put heavy constraints on the system, they make allocation predictable, immune from denial of service attacks, and allows ensuring a task will have enough resource to complete its execution.

We prove that building resource-secure systems is possible by describing the design and implementation of our prototype, Anaxagoros. The main issue for writing the system is synchronization, and we propose several novel ways to solve synchronization problems.

1 Introduction

A system that allows safe execution of hard real-time tasks is difficult to build. These tasks need prediction of the amount of CPU time and other resources necessary to complete their execution, and the only known way to achieve this is to strip down the system, removing most dynamic capabilities: no task creation, no dynamic extension of address space... Static allocation of resources is often the paradigm, because resource sharing is done using locks which makes schedulability analysis complex and pessimistic [56].

We believe that the fundamental reason why general purpose operating systems cannot safely execute hard real-time tasks is lack of *resource security*: it is difficult to guarantee that the system will give to a task the amount of resources that was planned. We think that strong resource security, combined with flexible allocation policies, would reconcile hard real-time with dynamic sharing, general purpose behavior.

This problem is especially important as the current trend in industrial control systems is to integrate several different functions in a single system, that were previously segregated into several systems [53]. Often these functions are of different importance, or *criticality*. They must be isolated from one another (and in particular the most critical ones must be protected), but at the same time the functions have to share the same set of resources: CPU time, memory, communication links, display... Stronger resource security would allow less conservative resource allocation and more resource sharing, increasing the system efficiency.

Contributions and plan Our first contribution is identification of the resource-security principles for efficient and secure support of hard real-time tasks with general-purpose tasks. These principles bring a highly secure system, yet dynamic and with flexible and precise resource management. Section 2 details these principles along with general design techniques to implement them.

Our main interrogation when we began designing the system was: “is it feasible to build a system that strictly follows these design principles?” Indeed building a secure system is known to be a hard work, and resource-security puts even heavier constraints: no unpredictable blocking, bounded synchronization time, no dynamic allocation of kernel/service memory, constant-time operations... Hence our second contribution is proving that building such a system is indeed feasible, and has a potential to be highly efficient. Section 3 details the design and implementation of Anaxagoros, a system built to strictly comply with those principles.

The main issue for building the system was synchronization, and our third contribution is a set of techniques for efficient solutions of synchronization problems that arise when building a resource-secure system, detailed in section 4. Section 5,6,7 provides an evaluation of the system so far, present related works and concludes.

2 Principles and global structure

Our goal is to build a system that safely and efficiently integrates hard real-time and non real-time tasks on the same system. This section first defines what would be a satisfactory integration, before giving the principles and general design rules to implement it.

2.1 Requirements and goal

The ideal system we want to build has the following characteristics:

1. It can safely share resources between the different tasks; for instance it allows tasks to share a network link or a graphical display. It allows "rich" OS operations such that dynamic task creation or dynamic extension or address space.

2. It makes impossible for a task to interfere with the correct execution of another task. In particular, a task being able to delay another is considered a security breach, as this could lead to a deadline miss.

3. It is simple for the system integrator to put the functions together. Adding a new task, resource or service does not need to reconsider the previous design. Assigning resources to tasks (and CPU time in particular) is simple, and not constrained by the particular system implementation.

Existing systems do not fulfill one or several of these characteristics. General purpose OS or hypervisors generally can block unexpectingly (e.g. when encountering a kernel semaphore or because of single-threaded services (e.g. [28, 62, 58]), and it is difficult to know the memory required for the operating system.

Hard real-time OS systems allow prediction of the number of resources to be used, and secure real-time OS use timing budgets to prevent delay by other tasks. But they often do not allow operations like dynamic task creation. Resource allocation is often static and sharing is rare, and done using locks that constrain to use a scheduling algorithm for which a schedulability analysis exists that can take them into account (in practice, the non-optimal fixed-priority algorithm).

To sum up, the system should provide facilities to safely share resources between tasks, i.e. using shared *services*. It should have *strong security* so as to prevent undesirable task interference, and security should cover protection of the hard real-time requirements of the tasks. The use of shared services by hard real-time tasks should be secured and easy.

2.2 General behavioral security

Fault containment (i.e. protection from interference) and traditional security have much in common [13, 53]: indeed, a breach in confidentiality also indicates that a task can affect another one. Therefore, spatial and behavioral security should be enforced through the systematic use of traditional security principles, as stated by Saltzer and Schroeder [55].

Separation of privilege States that the system should be divided into small parts, each with restricted privileges. To support this principle, our system defines three independent entities: *address space* (separates memory rights), *threads* (separates CPU access rights), and *domains* (separates all other kinds of rights). The usual *task* concept is obtained by juxtaposition of one thread, one domain, and one address space.

Least common mechanism System *services* are special-purpose code and data necessary to securely share resources between several tasks. Least common mechanism states that these services should be minimized, because they can cause more damage to the system (and also restricts its flexibility). There are two possible interpretations of this principle. Microkernel systems (e.g. [32, 57, 35]) minimize common mechanism by making tasks depend only on the services they use. Exokernel and hypervisor systems [17, 52, 5] do it by minimizing the size of these services. We chose to follow both interpretations in our system, by structuring the OS into small, simple, low-level and separated services.

Access control principles The principles state that each access to each resource should be systematically checked (complete mediation), that tasks should have no more privilege than required (least privilege), that access control should be based on list of permissions rather than list of bans (fail-safe defaults, also called closed-system design by Denning [13]), that the overall security mechanism should be simple (economy of mechanism), and that security of the system should not rely on secrecy (open design). We solve these issues by using *capability* [42] as the sole mechanism of access control: a domain can access an object only if it contains a capability to it. System capabilities implement all these principles, as they naturally implement a closed system, favor least privilege, are unforgeable and of simple design.

2.3 Real-time and resource security

The previous security principles provide a sound basis to deal with "behavioral" security (integrity, confidentiality, and spatial fault containment). But they are not sufficient

to protect from denial of service issues and temporal fault containment, especially in the case of real-time systems.

The main issue in hard real-time systems is "how to ensure that each job will have enough resources to execute before their deadlines ?". Although a part of the answer is related to scheduling and worst-case execution time research, there are two important system-related assumptions that must be made. The first is that when there is a chosen plan for resource allocation or schedule, resource allocation will follow that plan. This is what we call *resource security*. The second is that the amount of resources needed by the system can be predetermined.

2.3.1 Independence from allocation policies

Resource security means forbidding resource stealing, meaning that a task has fewer resource than was planned. A particular case is *denial of resource*, occurring when a pool of resource becomes empty and tasks that need it cannot complete their job.

These problems come from the fact that general purpose systems design does not require the resource allocation policy to be clearly defined, as resources are allocated dynamically according to demands. The amount of resources given to a particular task can unpredictably change, e.g. the kernel can page out a frame if it needs more memory, or the amount of CPU time given to a task is reduced because the OS encountered a semaphore. This makes them unsuitable for hard real-time tasks.

Therefore we defined the *independence from allocation policies* principle, which states that *resource allocation policies should be defined solely by a separated module*[1]. In other words, the OS no longer has the right to interfere with the chosen resource allocation, which is completely defined by the separated "policy module".

There are only a few ways by which the OS interacts with resource allocation, and we detail application of this principle for each.

Identifying and accounting for all resources Every overlooked resource creates a potential for denial of resource. For instance, if every pending request consumes unaccounted memory in a service, service memory can be exhausted with a sufficient number of requests. Some resources are "hidden": for instance address space for a service is limited, and clients that would need to insert memory mapping in the service address space would be rejected once address space is exhausted. Yet another example of easily exhaustible resources are TCP ports.

Most often denial of resource comes from a list-based allocation: e.g. a system call returns the first element of a free list (e.g. Solaris and Linux slabs [9, 10]). Instead, we systematically use *partitioning* for each kind of resource: each resource is *owned* by exactly one partition. Resource allocation is simply done by moving resources between partitions, and it is easy to account for the number of resources used in each partition.

Resource sharing is achieved by allowing several tasks to use the same partition, which is easily achieved by having one capability per partition. For instance, we have a common memory partition that contains the memory code for all libraries. This separation of permission and ownership combines the benefits of exact accounting of partitioning with flexible sharing of capabilities.

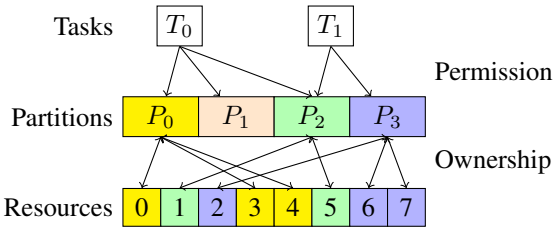

Figure 1: Separation of permission and ownership. Task T_0 can access P_0, P_1, and P_2, i.e. resources $0, 1, 3, 4, 5$.

No unexpected change in allocation Because the kernel is fully privileged, any piece of kernel code can change resource allocation. For instance, if the kernel needs memory, it can page out a process page to obtain it, changing the memory allocation. When it encounters a sleeplock, it can change previous scheduling decisions.

All this makes the allocation policy unpredictable and difficult to understand. Thus allocation policy decisions should be restricted to the module in charge (e.g. the memory allocator, the scheduler, etc.). In particular, it means that the kernel (or invocation of any service) should not block unexpectedly (only on explicit request), nor should it allocate memory on its behalf.

Independence of allocation from protection domain The system needs to be split into protection domains, and in particular we need separated services (or one kernel) to handle resource sharing and privileged operations. Thus tasks need to make requests (system or *service call*) to services. The problem is that these requests can consume resources themselves: if these resources are not correctly accounted for, this can cause a denial of resource.

The obvious way to solve this issue is to split the service resources in parts reserved for each client. For instance, the network service could reserve 3Mb of memory for client C's network buffers. There should be some of the service CPU time reserved to handle requests from C. While feasible, this approach has a number of issues:

- Reservation of resources (e.g. memory) in the service is a waste of resource if the resources are not

used. For many non real-time tasks it is difficult to know the amount of resources required.

- Static reservations of CPU time implies high client/request latency. Moreover, dividing the service CPU time between clients is difficult and error-prone.

- The approach cannot work when the number of clients is unknown, or can dynamically grow.

These issues are avoided if the clients provide the resources necessary to complete their own requests. For instance when doing a system call in a monolithic kernel, the CPU time spent in the kernel is accounted to the caller, and so is the memory used for stack (the process kernel stack). We call *lending* the act of providing to a service usage of a resource. Lending only takes place on the permission level, and does not affect ownership, and is thus independent from resource allocation policies. In the *thread lending* technique we explain on section 3.2, *all* the resources needed for a service call are provided by the client. This makes the system more efficient (no resources wasted), avoids denial of resource (no resource consumed by the service), and makes allocation easier (no need to reserve resource pools, crossing protection domain is independent from resource management).

Relation with policy/mechanism separation An interesting property of the independence from allocation policy principle is that the allocation policy for a resource is completely defined by the "policy module". Changing policies with the policy/mechanism separation principle [41] has thus more impact, because the "policy module" decisions are not constrained by the remainder of the system. For instance in our system it is easy to change the scheduling policy between round-robin, static scheduling, or even no scheduling (i.e. single-threaded system); the memory allocation policy between static and "first-come first-served".

2.3.2 Real-time and predictability

Predictable needed resources Supporting safety-critical real-time tasks requires the ability to predict the amount of resources a task will need to run correctly. This must be achieved through appropriate task design, but it is also needed to know the amount of resources consumed when using OS services.

Thus, the amount of *memory* consumed when using a service should be known to the client. In Anaxagoros, *all kernel objects are of size one page*, making their size easily predictable. Other services may document their needs in memory and other resources with their interface.

The amount of *CPU time* also has to be known. In our system *all kernel operations are done in constant time complexity* (even though we support complex operations like task creation, and multicore architectures). This is achieved by using low-level interfaces to services structured around the *resources automata*, which describes all possible operations based on the state of the resources.

Special attention is required to bound the time needed for synchronizations in multi-threaded services. In the kernel, the main other difficulty is bounding *object destruction*: forcefully retrieving resources in use must be a bounded operation, so as to timely restart a task. Object destruction is well-known as a difficult problem in capability systems [42, p.198].

Predictable errors It is not acceptable for critical tasks when a service request fails unpredictably.. Service calls should fail only because of incorrect use by the clients (wrong arguments, or incorrect call order). We found out that it was easy to achieve, once all denial of resources issues were avoided (with the independence from allocation policies principle).

3 System design and implementation

This section details the design and implementation of the Anaxagoros[2] microkernel, which is a building block for a secure system that comply with all the above principles. We first concentrate on access control and service call implementations, before explaining the memory system and other kernel services.

3.1 Design of the access control mechanism

3.1.1 Choosing an access control paradigm

System security requires a single ubiquitous access control mechanism. There are two main kinds of competing access control paradigms [37]: access control lists (ACL), and capabilities. We opted for capabilities mainly because:

- capabilities can perform access control checks in constant time (although some implementations do not [14]);

- the cost of access control storage can be attributed to the client, avoiding memory allocation in the kernel or services.

Capabilities have other advantages: they naturally implement closed systems, encourage least privilege, and fine-grained access control; they can implement a wide range of security policies [49].

3.1.2 Traditional capability implementation issues

But many capability *implementations* do not respect resource-security principles. One issue is *object destruction*: an object can be destroyed only if there are no more capabilities still pointing to it.

Existing approaches to the problem all suffer from different problems. Garbage collection [42, p. 198] allows an attacker to prevent the destruction of an object forever. Invalidating all capabilities pointing to an object one by one (e.g. seL4 [14]) can take an unpredictable and large amount of time if an attacker creates enough capabilities. The use of a unique identifier for objects (e.g. Hydra, System 38 [42, p.194], CAL/TSS [38] allows immediate invalidation of all outstanding capabilities to an object; but this identifier is stored in a central "master object table" [51, p.27] of limited size, and may thus be subject to denial of resource attacks.

A related problem is type destruction, which destroys at once all objects of a type (it is the same problem than object destruction, at a larger scale). Type destruction occurs for instance when a shared service is destroyed.

3.1.3 A capability system in constant time and space

We propose an efficient capability implementation, where all capability operations (invocation, creation, copy, object and type destruction) can be done in constant time complexity, and do not require a master object table. It is also very flexible and parallelizable.

Capability format and invocation Tasks *invoke* capability to an object so that the service responsible for this object can perform the required privileged operations[3]. But instead of performing the access control checks in one single operation in the kernel, we split access checks into three successive steps: checks for service access, checks for object access, checks rights. The kernel is only responsible for checking service access, but provides to the service the means to check object access and rights.

The capability structure contains a pointer to the service, the object number, and a set of rights to the object. The service pointer allows the kernel to retrieve the service in constant time. The object number allows the service to retrieve the object in constant time. The capability structure contains a fourth field, called the *timestamp*, that contains the "creation date" of the capability. Services and objects also have a timestamp: the object timestamp is stored in a per-service object table, and the service in the *frame table* (a table with one entry per physical frame, see section 3.3.5). Thus there are no central tables, and the timestamps are easily retrieved from the service pointer or object number.

Access checks Intuitively, access should be granted only if neither the object and the service have been destroyed since creation of the object/capability pair. This is what the timestamps algorithm implements (Figure 2): timestamps monitor capability, service and object creation. Access is denied iff the service and object were created after the capability (it means that the storage has been reused and the capability is invalid). Destruction is represented as creation of an object with timestamp $+\infty$. Thus, capabilities identifies services and objects uniquely, both spatially and temporally.

The set of rights in a capability is represented by a bitfield, with one bit set per right owned. Checking that rights is sufficient is a simple mask operation.

The access checks need to be done upon capability invocation, but not only: because services calls can be done in parallel, objects and services can be destroyed while a thread is performing operations on it. On single processor, it is sufficient to add checks when a thread resumes execution in the service. On multiple processor, interprocessor interrupts must be added to throw threads of other cores out of the service (see section 4.1.2).

Summary The timestamp algorithm has many advantages: small constant-time bounds on object access checks, object destruction, and service destruction; no central array, and services use their own storage for their object table; compact capability representation. Finally, it is possible to write a highly parallel version of this implementation for multicore systems. This parallel version and the proof of its correctness are described in [39]. More details can also be found in [40].

3.2 Service call: the thread lending model

Access control is only a part of the service call mechanism. We now deal with data and resource transfers, centered in our system around the resource lending mecha-

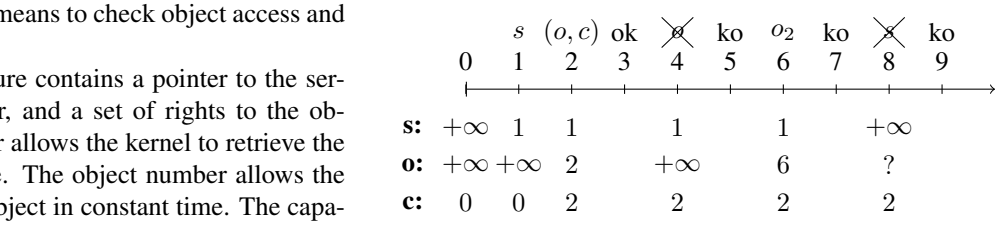

Figure 2: Evolution of timestamps for a service s, object o, and capability c. The three bottom lines are timestamps of s, o and c. The top line represent events: "ok"and "ko" represent sucessful and denied invocations of c. Event 1,2 and 6 are creation of s, o, c and o_2 which occupies the same storage than o. Event 4 and 8 are destruction of o and s.

nism. We define *resource lending* as the transfer of the right to use a resource, without changing ownership of the resource. In other words, a service can use the resources provided by a client to complete a request, but these resources still belong to the client, are accounted to him, and resource allocation does not change. This general principle corresponds to different realities depending on the kind of resource.

Lending CPU time Applying resource lending to CPU time means that *execution of the client request in the service happens when the client should have run*. This can be achieved using a "thread tunnelling" [52] mechanism: the client thread continues its execution, but in the service protection domain. System calls to a monolithic kernel is an example of such a system, but there also exists many implementations of the technique for userspace services (e.g. [63, 25, 16, 22]). This mechanism allows low-overhead, low-latency client-service communication like synchronous IPC [43, 57], but without interfering with the scheduler.

In particular the client thread can be preempted while it is in the service. Thus shared services become by construction multi-threaded, with one thread per current client connection. All these threads consume memory for their stack, and the number of clients may not be *a priori* bounded, so stack memory must also be lent by the client to avoid a denial of resource vulnerability.

Problems of lending capabilities and memory Memory and capabilities are very similar: indeed in both case a reference to a resource (resp. the page table entry and the capability), stored in a table (resp. a page table or capability table) gives the right to access a resource. The natural way (e.g. [44]) to lend the resource is thus to copy the reference in the corresponding table in the service. But service tables are finite, so this constitutes a possible denial-of-resource attack on the "service table entry" resource. To solve this problem, the client must lend the room for table entry as well. And it cannot just lend memory for this, as this constitutes a chicken-and-egg problem.

Thread lending To solve this problem, Anaxagoros threads are also *principals*, i.e. they can be used to hold references to resources: there are *thread-local capabilities*, and *thread-local mappings*. Program execution can use the memory of the current address space or the current thread, and the capabilities of the current domain or the current thread.

When a thread is lent, it means that all of its CPU time, thread-local capabilities, and thread-local mappings are lent. Concretely, the client prepares the thread with the mappings and capabilities it wants to lend, before passing the thread to the service. Notice the analogy with the implementation of "passive call" in object-oriented languages, with access control metadata being exchanged by domains using threads (instead of data being exchanged by objects using the stack).

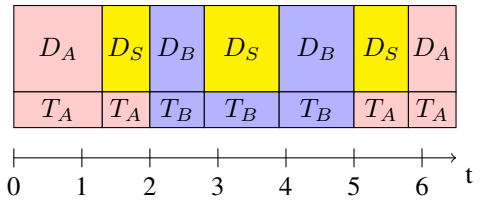

Figure 3: Evolution of thread (T_*) and domain (D_*) mappings for two tasks A and B calling a service S. Preemption (at time 2 and 5) change the current thread; service calls/return change the current thread's domain.

Implementation The "threads" kernel object contains a pointer to the "current domain" object, which contains a pointer to the "current address space" object. After the kernel has checked that the capability can access the service (section 3.1.3), the thread's current domain changes to the service, and the address space changes to that of the domain. The return is similar.

Our first implementation of thread-local mappings copied the thread-local mappings to a reserved location in the service page directory, upon the service call and when threads resume from preemption (see Figure 3). There were some TLB issues on multiple processors, and our new design now copies the service and the thread mappings on a per-processor page directory. These operations are necessary because of the hardware page tables, and would be much easier to accomplish with a software TLB.

There is a special mapping, the *UTCB*, which is guaranteed to be mapped only in the current thread, so that it can safely be used as stack by the service. It has other uses, such as passing arguments to services, or asynchronous communication with the kernel (section 4.1.1).

Discussion and related work The thread lending service call model has many advantages. It retains the low-latency and low-overhead of the synchronous IPC model [43, 58] without the need to block, which makes schedulability analysis easier, and allows parallel service execution in the service on multicore systems. It also avoids the denial of resource issues found in many thread-tunnelling implementations (e.g. shortage of server thread in Spring [25]).

The biggest benefit of the model is that it does not change any resource allocation policies. This makes

lending fast because resource allocation modules (e.g. scheduler or memory allocators) do not need to be involved in the service call. It also makes policies simpler.

But thread lending is no silver bullet. In particular thread lending alone is difficult to use for requests that may not be immediately satisfied, for instance disk reads or network requests. In this case there should be a service thread to handle, schedule and serialize the requests. However, thread lending is still useful to set up and communicate with this service thread. For instance, it can be used to set up memory lending that lasts across service calls, in the spirit of EROS network stack [60].

Finally, thread lending causes many synchronization issues in shared services, that are addressed in section 4.

3.3 The memory management service

Memory management is another key piece in achieving behavioral security, as it is responsible for ensuring confinement of memory accesses. Because of its critical role, the memory management service is part of the Anaxagoros kernel.

The current implementation is for the Intel x86 architecture, however it should be readily portable to any architecture with multilevel paging. Its complete description, and full proof of confidentiality of the system (and of its liveness) is available [39]. More detailed descriptions can also be found in [40].

3.3.1 Kernel services interface

The virtual memory service provides a low-level interface: clients must select the individual frames and sequence of privileged operations that they want to be done on them. Higher level functionality (e.g. task creation, address space extension) can be provided either by libraries [17, 34] or virtualization [5].

The service is centered around the *frame type automaton*, which describes the different roles that can be assumed by a frame and the transition between them. For instance, a memory frame can be used to hold regular data, but must be entirely cleaned of this data before being used as a page table. The current type of a frame restrict the operations that can be performed on them: for instance data mappings can only be installed on page tables, and to data frames.

The kernel only role is to make sure that the operations are valid according to the frame type automaton, and are allowed according to ownership: privileged operations on a frame can be done only by clients that have a capability to the partition that contains it[4].

3.3.2 Frame types description

There are few different frame types:

Dataframe This type holds regular data, and (with UTCB) is the only type directly accessible from user space.

Page table and page directory These are the page table of the different levels. They only contain page table entries to the lower-level page table, or to dataframes. The page directory is the top-level page table, and represents the "address space" kernel object.

KTCB and KDCB Hold and represent respectively the "thread" and "domain" kernel objects (the acronyms stand for kernel thread control block and kernel domain control block). They both contain capabilities, as well as other (e.g. scheduling-related) data.

UTCB The user thread control block is uniquely associated to its KTCB. It is used as storage for client-service and client-kernel communication. It is writable, but is mapped only once, in the current address of the thread, so that it can safely be used as a stack by the service.

Zero and cleanup states These states are necessary intermediaries for a frame to change type. The zero type is an unmapped page filled with zeros. When a frame changes to one of the above frame types, it does so by a single transition from the zero state. The other types represent intermediary cleanup states: before a frame can be reused, it must be cleaned to return to the zero state. The intermediary states are used to record that only a fraction of the frame has been cleaned up, which allows splitting the cleanup operation of frame, decreasing the preemption delay and allowing low-latency task switches.

3.3.3 Memory mappings

An important privileged operation is page table modification. When the kernel creates new page table entries, it must obviously ensure that the pointed page is a lower-level page table (or data frame), otherwise this would result in an integrity breach.

But it must also ensure that when frames change to another type, there are no existing page tables that point to them. Although the timestamp mechanism could be used to solve this problem, this would incur a large memory overhead. Instead, we use a *reference count* [5], counting how many times a page is present in a higher-level page table. This count is updated whenever page table entries change, and must be equal to zero before the frame can be cleaned.

3.3.4 Consistency issues and multicore

On single processor, our kernel is atomic and follow the interrupt model [19], which made implementation of the virtual memory system relatively easy. To keep task switching latency low, the code explicitly polls to know

if there is a preemption pending, and if so clean up and perform the task switch. This makes code much simpler than pessimistically being prepared for preemptions, and regular polling still allows for low latency.

The only consistency problem is enforcing TLB consistency. This is done by the kernel (by flushing the TLB) only when this would otherwise cause a security problem (other flushes are required by the clients). This allows user-space to amortize the cost of a flush between multiple page-table modifications.

We realized an implementation of the virtual memory system on multiple processors (but did not yet integrate it to the kernel). Our implementation relies on a number of "partial guarantees" [27] techniques. One is kernel atomicity: operations are ensured to terminate. One is the *use/destroy* lock (section 4.1.2): concurrent reads and modifications of an object are allowed, but the lock forbids access to an object that is being destroyed by another processor. Thus, only the few conflicting operations are forbidden, and all other accesses can be done in parallel.

Parallel modifications are handled using various ad-hoc wait-free techniques: for instance, checking a frame state and changing it can be done in a single instruction using compare and swap. Another technique is out-of-sync reference counting: reference count can be greater than the actual number of references, which avoids synchronization between these values. The result is that our implementation never requires busy-waiting, except (the rare) forceful kernel object destruction for which busy-waiting is bounded. This makes our implementation highly parallel, with all operations bounded in time, even on multicore systems.

We found that the low-level interface, and decomposition in small actions allowed for highly parallel implementation. These techniques can be generalized for user-level services, as shown in section 4.2.

3.3.5 Experience designing the memory service

Because of the complex relationship that can exist between the frames, the virtual memory system is the most complex part of the kernel. It is thus a good test for applying the resource-security principles.

Its interface has many advantages. It is easy to predict the amount of memory needed for kernel objects, as they all are of size one page. All operations take a bounded time, and thus system time can be easily accounted for. The *frame table*, which contains an entry per frame, is the only necessary storage for all of its operation, and there is no need for dynamic memory allocation in the kernel (the frame table is allocated at initialization time). The frame table is also used to store the services timestamps, and thus integrates well with the capability system.

The experience building the virtual memory system has been central for determining the applicability of our principles, and providing design guidelines to respect them. This was true in particular for synchronization issues, discussed below.

4 Synchronization issues in shared services

The most difficult problem we encountered when writing resource-secure code is synchronization, due to both the constraints of the resource-security principles and the concurrent nature of shared system services. There are two kinds of responses to this problem: coding techniques, and design methodology.

4.1 Dealing with forceful revocation

Resource lending means that service code use resources they do not "own", and their right to use the resources can be revoked anytime (e.g. when the client is destroyed). Resource-security principles forbid notifying the service and waiting for it to release the resource (as in [18]), as it is difficult to account for the "extra time" needed to handle these notifications. Instead, services must be prepared to sudden revocation of these resources.

Revocation of memory and capabilities can be easily handled. If access to a resource is revoked, invoking its capability will simply report an error. Accessing a revoked memory region can be handled using an exception-like procedure implemented with self-paging [15, 26] or user-level pagers [1, 28].

In fact, when access to a revoked resource is tried by a lent thread, most often the best way to handle this is to return to the client with an error code. This can be seen as a special case of preemption with infinite duration. Thus the service only has to deal with preemption, i.e. revocation of CPU time.

4.1.1 Revocation of CPU time and preemption

A notable difference between shared services and conventional multithreaded programs is that the service has no control over when the lent threads are run. For instance, a scheduling policy can preempt a lent thread inside a service, and never execute it again. Furthermore, as services are forbidden to affect scheduling by blocking, they cannot use conventional facilities such as semaphores or sleeplocks (locks that put tasks to sleep).

Because services are multithreaded and access shared state, some kind of synchronization is however necessary. An alternative to sleeplocks is lock-free programming. We heavily used it, but we found that "general"

lock-free algorithms [29, 30, 21] require unbounded allocation of the service memory. Another one is hardware transactional memory [31], but is currently available only on a few platforms.

Roll-forward locking The last alternative to sleeplocks are spinlocks, but they cannot be used because preemption in the critical section would make other threads to spin forever. The classical solution to this problem is to mask interrupts, but this is a privileged instruction unavailable to our userspace services. Another solution [16, 47] is to give tasks an "extra time" when they are in critical section, but that would increase the task switching latency, which is undesirable for hard real-time systems[5].

Instead, our technique has been to allow for *recovery*: when a thread needs a lock used by a thread that was preempted in its critical section, it releases the lock by terminating execution of the critical section in place of the preempted thread. It can then take the lock to execute its own critical section. This recovery strategy is called *roll-forward* [6] .

Implementation The mechanism is based upon another mechanism we call *user-level preemption and resuming*: upon preemption, the kernel writes all registers in user-space at a location indicated by the thread. Upon resumption, the thread is responsible for restoring its context by itself. This mechanism is similar to those in [3, 17].

When a thread acquires the lock, it indicates to the kernel the address of the lock, and switch to a dedicated stack in memory owned by the service. If it is preempted, all registers are stored on top of this stack, and the kernel changes the value of the lock to indicate preemption to other threads. If another thread needs the lock, it restores the registers and continue execution until the lock is released.

The implementation does not need to perform any system call, which makes it very efficient, but synchronization issues make it an extremely complex piece of assembly code. Fortunately this complexity is hidden by a convenient API.

Other recovery mechanisms A drawback of roll-forward is that the critical section cannot access the current thread-local mappings (because execution can be done using another thread), which limits its applicability. For this reason we are considering other recovery strategies.

Rollback can be implemented by writing a back-log of the previous values of the stores that are done. But it is not applicable to some device drivers (e.g. VGA display), for which special registers have to be written in order. We are also considering allowing "ad-hoc" recovery strategies for performance-critical cases: this can be seen as an extension[6] to the concept of "revocable" lock [27].

4.1.2 The use/destroy synchronization protocol

The previous section dealt with revocation of resources lent. We now focus on destruction of the objects served by the service (i.e. revocation of the resources used for this object).

When an object is destroyed (e.g. a network connection is forcefully destroyed), eventually no more thread should be operating on the object. This must happen before the resources of the object (e.g. the memory for the network buffers) can be reused. Moreover, to comply with the predictability principle, the time spent waiting for possible reuse of an object should be bounded.

To solve this problem, we use the following protocol:

Using an object:
1. If object destroyed: leave
2. Mark the object as being used
3. If object destroyed: release the object
4. Operate on the object
5. Notification received or polling: release the object

Destroying an object:
1. Mark the object as destroyed
2. Notify all running user threads of object destruction
3. Busy-wait until all user threads are gone
4. Mark the object as reusable

Upon destruction, step 1 prevents new user threads from coming in, while step 2 urges currently running (on other processors) user threads to stop using the object. After a while, no thread is using the object, and it can be safely reused. The step 3 when using an object is necessary to avoid a race condition.

Implementations of the protocol We have several uses of this protocol. The virtual memory service use it before cleaning page types. For pagetables, no notification is necessary, because the code regularly polls to see if the page has been destroyed. For other types (domain, thread, address space destruction), an inter-processor interrupt is sent to processors using the object so that they return to the kernel.

In user-level services, destruction of an object is done by changing the timestamp of the object. This prevents new threads from using the object, as well as preempted threads to reuse the object when they resume . On multiprocessor, an interprocessor interrupt could be sent to

other threads of the service so that they stop using the object.

Note that in single processor systems, steps 2 and 3 are not necessary, because there cannot be any concurrent use when the resource is destroyed. Steps 1 and 4 can thus be done in a single operation.

Discussion This technique allows timely destruction of an object (i.e. revocation of a resource), without burden on the rest of the code. Timely destruction only require user threads to timely stop using the resource once they have been notified, which is immediate when using inter-processor interrupts, and fast when polling regularly. Marking the resource as being used is done simply with some kind of reference counting. Multiple threads can be using the same object concurrently.

The use/destroy lock provides "partial guarantees" [27]. For instance, as long as the object is used, its storage cannot be reused, providing "type-stability" [23]. We found that this kind of partial guarantees is sufficient to make design of wait-free algorithms tractable.

4.2 Designing resource-secure services

Even if using roll-forward locks does not affect scheduling, they can induce a variation in execution time. So as to keep this variation to a minimum, synchronizations should be kept at a minimum. The following design rules helped us to write various services with minimum synchronizations, and that comply with the resource-security principles. In one word, the motto is *minimization*.

4.2.1 Minimization of state

The least state in a service, the least data to synchronize. There are different techniques to minimize state: one of the most important is to suppress abstractions from the service [17, 34] (and provide abstraction in libraries). This structures the service around a *resource array*, with one entry per physical resource[7], and a few global variables. Another technique is to make transactions stateless, i.e. pass data as arguments rather than retain it in the service.

Following this principle is interesting for resource-security, but also for traditional security (least common mechanism), for multicore performance (less data shared), and flexibility and performance in general (as shown by [17, 52, 34])

4.2.2 Minimization of actions

The second principle asks to design the service interface around a set of small orthogonal actions. Instead of providing complex operations such as `mmap` or `writev`, it is better to structure the system around basic operations such as "clean page table entry" or "putchar". This makes critical sections short and fine-grained (thus more easily replaceable by wait-free or lock-free algorithms); allows better context-switching latency in the kernel, and less time spent in recovery in user services.

However, division into small actions can be inefficient because of the overhead of "setting up" the action (i.e. context switch, taking a lock, etc.). The interface should provide a way to group the actions efficiently (for instance, in the memory service we allow to set up "mapping ranges" to mutualize the syscall and "use lock" overheads, but this operation can still be stopped at the granularity of writing one page table entry).

Generally, minimization of actions means that the service is structured around *resource automata*, as in our memory system or in [54]. Often the automata is simple, with only one reinitialization phase and one operational phase.

4.2.3 Minimization of synchronizations

We observed that in many cases, the fact that the remaining data managed by the service can be inconsistent is not a security problem. For instance, several threads writing simultaneously to the same network buffer will likely send garbage, but will not prevent other threads to send proper data.

Thus whenever possible, *we make clients responsible for the consistency in the service data.* In fact, this is something existing OSes must already do. For instance in the network buffer case, serializing the writes in a service using a mutex would not be sufficient, because the contents of the network data depends on the order of the writes, that an OS service cannot control. We only make this fact explicit.

Ensuring consistency at the client level is not difficult. Most often a policy will ensure that different clients will access different resources. When multiple clients access the same resource, they generally need to synchronize anyway, because operations to the resource have to be done in a certain order. An exception to this rule is forceful destruction and retrieval of a resource, which must succeed regardless of concurrent operations on the resource. This is the purpose of the use/destroy lock.

Even when consistency has to be ensured for security reasons, it is often not necessary to enforce it through serialization of requests. It is often easier to detect and report inconsistency as an error. For instance rather than serializing the writes to a capability table entry , we detect concurrent writes to the same entry, and report an error. Early detection of errors prevents further propagation, and is important for fault-tolerant systems [13].

Another example where this principle applies is ensuring TLB consistency in our virtual memory system. The TLB can hold references to entries not present in page tables, but only as long as this is not a security threat (i.e. this may not be used to write to kernel objects).

5 Evaluation

Feasibility and experience We have obtained a first prototype that fully respects resource-security (no blocking, no service dynamic memory allocation), behavioral security (separated, minimal user-level services), and maximizes resource lending. It comprises several user-level resource-secure services: textual VGA display, keyboard, a ram file system, the beginning of a network stack, and several kernel-level services (thread and scheduling management, domain and protection, virtual memory and I/O ports). It also comprise a "libOS", especially used for memory management (memory map and allocation, creating new tasks from ELF images, etc.). The system allows using shared services like the VGA display and dynamic creation of tasks without any impact on scheduling or memory allocation (which can be static). This shows that allocation policies is independent from the use of the resource.

Implementing this prototype helped us more clearly define the resource-security concepts given in the paper, and find out the techniques necessary to overcome the constraints. The most complex issue during implementation of the system was synchronization. Especially, the combination of multithreaded services (due to tread lending), no sleeplock (because of resource-security principles) and no spinlock with interrupts masked (because of security principles) forced use to explore new solutions, like the roll-forward lock. Writing parallel services is not hard, once we have a clear view of the requirements and concurrent operations involved.

Security It is difficult to provide benchmark for security. An important metric is size, because fewer code means fewer bugs. Our kernel currently has 2282 lines of C and 1088 of x86 assembly (measured with sloccount). A large part of them (500 statements) deals with verifying user input and internal assertions. The resulting kernel code is less than 60kb (this could be further decreased after optimizations). Many services are much smaller: for instance the user-level VGA display service code fits in one 4kb page.

A goal of our kernel is to efficiently support multicore systems, and we designed the kernel with these systems in mind. We are in the process of rewriting the kernel to support these systems, and found that parallel code may be complex to understand (especially in the virtual memory service). This is why we did a full manual proof of these algorithms [39], and began formally specifying some parts with TLA [36]. We found out that proof allowed to fully understand the precise requirements of the algorithms, and to minimize the amount of synchronization needed to fulfill these requirements (i.e. write potentially more parallel code).

Performance The focus for this initial prototype has been put first on security, second on simplicity, but even before the optimization phase many operations already have correct performance. The following measurements were performed using the `rdtsc` instruction (which reads the number of cycles) on a Athlon XP 3000+ processor (first line) and the bochs PC simulator (second line. bochs does not simulate cache, and execute one instruction per cycle).

call	new pd	new pt	new dom	free pd	vga
4687	577	233	1879	44156	259279
750	160	120	445	31755	35570

This array gives the number of cycles needed to do a service call to the VGA service to write a string of one character; create a new page directory, page table, domain; to remove all entries in a page directory; and to set up the VGA service (i.e. create its page table and directory from the ELF binary loaded into RAM, call it so that it can initialize, and return). Most of these operations are quite fast. Destruction of page directory is long because it requires to remove all the mappings. Creating a new service needs a lot of service calls for now, and could be optimized a lot, for intance by batching system calls to memory. In a previous experiment, we found that service call time can be reduced to 1500 cycles when all of the physical memory is accessible to the kernel (else, this requires the kernel to set up expensive temporary mappings).

There are several reasons why we expect to get performant in the future. The non-blocking property means less time lost in the scheduler and re-filling the caches. Services are well-suited for multicore execution, because they are multithreaded and with minimal shared state. Resource lending means less resources wasted in static reservations.

Cache effects We did some experiments to measure the variation of actual execution time. The threads were statically scheduled with 10ms timeslices. Threads performed various workload (service calls, filling the cache...), and one thread incremented a counter. The value of the counter was compared after each timeslice.

In the worst case, the execution variation measured with bochs was of 1200 cycles; with the Athlon XP 3000+ processor it could reach 150000 cycles. This

means that resource security should be complemented with an approach to partition the cache, for instance page coloring [45]. But resource security already helps control cache unpredictability: because threads cannot block unexpectingly, possible preemption instants (i.e. possible cache flushes) can be limited. In the example, there cannot be more than one preemption every 10ms, and the variation in execution time was below 1%.

6 Related work

Many systems have been built to improve resource accounting and security on general purpose systems, but generally to support soft real-time and multimedia tasks, not safety-critical hard real-time.

Nemesis [52] analyzed that using shared services can cause some CPU time to be unaccounted for, and proposed to minimize this unaccounted time by minimizing the service. Thread lending allows for exact accounting of CPU time in shared services, but still recommends their minimization.

Other approaches allowed accounting of CPU time spent in shared services: capacity reserves on microkernels [48, 62], resource containers on monolithic kernels [4]. An important difference with resource-security principles is that they require not only correct accounting, but also not to affect scheduling decisions.

Other systems were built to avoid denial of resource, and especially for memory. KeyKOS [8], EROS [59] and the Cache kernel [12] avoid kernel memory allocation by viewing kernel memory as a cache, which is not suitable for real-time systems.

Liedtke advocated for the benefits of memory lending against denial of service attacks [46]. Genode/Bastei implemented a mechanism of temporary resource donation [18], (different from lending because memory allocation changes). L4 [24], seL4 [14] and Xen [5] have implemented memory lending, but only to the kernel/hypervisor. CAP [50] and EROS [60] did implement memory lending to any shared services, but not systematically in each communication.

The thread-tunneling mechanism is common [7, 25, 20, 17]but without lending of stack is generally vulnerable to denial-of-resource on kernel memory (i.e. unpredictable blocking). An exception is the Pebble mechanism [22], which can allocate lend a stack.

As single-threaded services are problematic for real-time and multicore processing, multithreaded services have been advocated for L4 [28] and Nova [61], but without memory lending would lead to higher memory consumption.

Rushby [53] and MILS systems [2] propose a partitioning approach relying on static allocation which is resource-secure. We think resource-security is possible with more dynamic behavior, which increases performance and allows support for general-purpose tasks.

There as been many scheduling-related work on supporting real-time applications on general-purpose OSes (e.g. [33, 11]). These approaches are complementary to resource-security.

Finally, our design and implementation was inspired by reading many techniques in other non-blocking systems [23], other systems with low-level interfaces [52, 5, 17], other microkernels [43] and capability systems [59, 8, 63, 42, 38, 51].

7 Conclusion

In this paper we explained how resource-security principles are necessary to safely execute hard real-time and general purpose tasks on the same system. These principles allows to predict when a task will have enough resources to execute, allows for exact, flexible resource accounting, encourage high resource sharing, and makes definition of resource allocation easier.

We explained how we solved design issues when implementing a operating system microkernel that comply with these principles. We showed how many synchronization problems encountered when using shared services needed new solutions explained in the paper.

Our prototype kernel proved that applying the resource-security principles is possible. But a lot of work still has to be done to fully demonstrate the advantages of a full resource-secure systems.

More shared services should be written. Work has began on a network stack implementation, which is a good example of a complex service that could be compared to other systems. A problem is that getting resource-secure services require redeveloping it, and synchronization issues in shared services are hard, so it would be interesting to provide libraries (or driver synthesis [54]) to simplify the service development process.

The system has to be optimized for full performance evaluation of resource security. Resource-security principles should be beneficial to multicore systems (because shared state and synchronizations are minimized), so scalability should be taken into account. We designed the kernel with multicore systems in mind, and already began re-implementing some parts.

Finally, even if resource security allows almost perfect resource allocation, current hardware is not optimized for the worst case and make it easy for a task to affect the performance of another task (e.g. with cache pollution). We should investigate solutions to this problem; for instance it might be possible to partition caches using page coloring [45], or limiting the number of preemptions.

References

[1] ACCETTA, M., BARON, R., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. Mach: A new kernel foundation for unix development. Tech. rep., Carnegie Mellon University, August 1986.

[2] ALVES-FOSS, J., HARRISON, W. S., OMAN, P., AND TAYLOR, C. The mils architecture for high-assurance embedded systems. *International journal of embedded systems ISSN 1741-1068 2*, 3-4 (2006), 239–247.

[3] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst. 10*, 1 (1992), 53–79.

[4] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Proceedings of OSDI '99* (1999), USENIX, pp. 45–58.

[5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of ACM SOSP '03:* (2003), ACM Press, pp. 164–177.

[6] BERSHAD, B. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems* (May 1993), pp. 264–273.

[7] BERSHAD, B., ANDERSON, T., LAZOWSKA, E., AND LEVY, H. Lightweight remote procedure call. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles* (New York, NY, USA, 1989), ACM Press, pp. 102–113.

[8] BOMBERGER, A. C., FRANTZ, A. P., FRANTZ, W. S., HARDY, A. C., HARDY, N. R., LANDAU, C., AND SHAPIRO, J. The keykos nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (April 1992), pp. 95–112.

[9] BONWICK, J. The slab allocator: an object-caching kernel memory allocator. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference* (Berkeley, CA, USA, 1994), USENIX Association, pp. 6–6.

[10] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel - 3rd edition*. O'reilly, 2005.

[11] BRANDT, S. A., BANACHOWSKI, S., LIN, C., AND BISSON, T. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium* (Washington, DC, USA, 2003), IEEE Computer Society, p. 396.

[12] CHERITON, D. R., AND DUDA, K. J. A caching model of operating system kernel functionality. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 1994), USENIX Association, pp. 179–193.

[13] DENNING, P. J. Fault tolerant operating systems. *ACM Computing Survey 8*, 4 (1976), 359–389.

[14] ELKADUWE, D., DERRIN, P., AND ELPHINSTONE, K. Kernel design for isolation and assurance of physical memory. In *1st Workshop on Isolation and Integration in Embedded Systems (IIES'08), Glasgow, UK* (April 2008).

[15] ENGLER, D., GUPTA, S., AND KAASHOEK, M. AVM: application-level virtual memory. In *Proceedings Fifth Workshop on Hot Topics in Operating Systems* (May 1995), pp. 72–77.

[16] ENGLER, D. R. The design and implementation of a prototype exokernel system. Master's thesis, Massachussets Institute of Technology, 1995.

[17] ENGLER, D. R., KAASHOEK, M. F., AND J. O'TOOLE, J. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of SOSP '95* (1995), ACM Press, pp. 251–266.

[18] FESKE, N., AND HELMUTH, C. Design of the bastei os architecture. Tech. Rep. TUD-FI06-07, Technische Universität Dresden, December 2006.

[19] FORD, B., HIBLER, M., LEPREAU, J., MCGRATH, R., AND TULLMANN, P. Interface and execution models in the fluke kernel. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation* (Berkeley, CA, USA, 1999), USENIX Association, pp. 101–115.

[20] FORD, B., AND LEPREAU, J. Evolving Mach 3.0 to a migrating thread model. In *Usenix Winter Conference* (1994), pp. 97–114.

[21] FRASER, K. Practical lock-freedom. Tech. rep., University of Cambridge, February 2004.

[22] GABBER, E., SMALL, C., BRUNO, J., BRUSTOLONI, J., AND SILBERSCHATZ, A. The Pebble component-based operating system. In *Proceedings of the 1999 USENIX Technical Conference* (June 1999), pp. 267–282.

[23] GREENWALD, M., AND CHERITON, D. R. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation* (1996), pp. 123–136.

[24] HAEBERLEN, A., AND ELPHINSTONE, K. User-level management of kernel memory. In *Proceedings of the 8th Asia-Pacific Computer Systems Architecture Conference* (Aizu-Wakamatsu City, Japan, Sept. 24–26 2003).

[25] HAMILTON, G., AND KOUGIOURIS, P. The Spring nucleus: A microkernel for objects. Tech. Rep. TR-93-14, Sun Microsystems Laboratories, Inc, April 1993.

[26] HAND, S. M. Self-paging in the nemesis operating system. In *Operating Systems Design and Implementation* (1999), pp. 73–86.

[27] HARRIS, T., AND FRASER, K. Revocable locks for non-blocking programming. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2005), ACM, pp. 72–82.

[28] HÄRTIG, H., HOHMUTH, M., LIEDTKE, J., SCHÖNBERG, S., AND WOLTER, J. The performance of μ-kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), ACM Press, pp. 66–77.

[29] HERLIHY, M. A methodology for implementing highly concurrent data structures. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming* (New York, NY, USA, 1990), ACM, pp. 197–206.

[30] HERLIHY, M. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems 15*, 5 (November 1993), 745–770.

[31] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture* (New York, NY, USA, 1993), ACM, pp. 289–300.

[32] HOHMUTH, M., PETER, M., HÄRTIG, H., AND SHAPIRO, J. S. Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors. In *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2004), ACM, p. 22.

[33] JONES, M. B., ROSU, D., AND ROSU, M.-C. CPU reservations and time constraints: efficient, predictable scheduling of independent activities. *SIGOPS Oper. Syst. Rev. 31*, 5 (1997), 198–211.

[34] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., BRICENO, H. M., HUNT, R., MAZIERES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on exokernel systems. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1997), ACM Press, pp. 52–65.

[35] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT, USA, Oct 2009), ACM.

[36] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16*, 3 (1994), 923.

[37] LAMPSON, B. Protection. *ACM Operating System Review 1* (January 1971), 18–24.

[38] LAMPSON, B. W., AND STURGIS, H. E. Reflections on an operating system design. *Commun. ACM 19*, 5 (1976), 251–265.

[39] LEMERRE, M. *Intégration de systèmes hétérogènes en terme de niveaux de sécurité*. PhD thesis, Université Paris Sud, October 2009.

[40] LEMERRE, M., DAVID, V., AND VIDAL-NAQUET, G. A dependable kernel design for resource isolation and protection. In *IIDS '10: Proceedings of the First Workshop on Isolation and Integration in Dependable Systems* (2010), ACM, pp. 1–6.

[41] LEVIN, R., COHEN, E., CORWIN, W., POLLACK, F., AND WULF, W. Policy/mechanism separation in Hydra. In *Proceedings of SOSP '75* (New York, NY, USA, 1975), ACM, pp. 132–140.

[42] LEVY, H. M. *Capability-Based Computer Systems*. Digital Press, 1984.

[43] LIEDTKE, J. Improving IPC by kernel design. In *Proceedings of SOSP'93* (Asheville, NC, Dec. 1993).

[44] LIEDTKE, J. On micro-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1995), ACM Press, pp. 237–250.

[45] LIEDTKE, J., HÄRTIG, H., AND HOHMUTH, M. Os-controlled cache predictability. In *Proceedings of the 3rs IEEE Real-time Technology and Applications Symposium (RTAS)* (Montreal, Canada, June 1997).

[46] LIEDTKE, J., ISLAM, N., AND JAEGER, T. Preventing denial-of-service attacks on a microkernel for weboses. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)* (Cape Cod, MA, May 5–6 1997).

[47] MARSH, B. D., SCOTT, M. L., LEBLANC, T. J., AND MARKATOS, E. P. First-class user-level threads. In *Proceedings of the13th ACM Symposium on Operating Systems Principle* (Pacific Grove, CA, 1991), pp. 110–121. Psyche.

[48] MERCER, C. W., SAVAGE, S., AND TOKUDA, H. Processor capacity reserves for multimedia operating systems. Tech. Rep. CS-93-157, Carnegie Mellon University, 1993.

[49] MILLER, M., AND SHAPIRO, J. Paradigm regained: Abstraction mechanism for access control, 2003.

[50] NEEDHAM, R. M., AND WALKER, R. D. The cambridge cap computer and its protection system. In *SOSP '77: Proceedings of the sixth ACM symposium on Operating systems principles* (New York, NY, USA, 1977), ACM Press, pp. 1–10.

[51] REDELL, D. *Naming and protection in extendable operating systems*. PhD thesis, MIT, 1974.

[52] ROSCOE, T. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge, April 1995.

[53] RUSHBY, J. Partitioning in avionics architectures: Requirements, 1998.

[54] RYZHYK, L., CHUBB, P., KUZ, I., SUEUR, E. L., AND HEISER, G. Automatic device driver synthesis with termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)* (Big Sky, MT, USA, October 2009).

[55] SALTZER, AND SCHROEDER. The protection of information in computer systems. *Communication of the ACM 7* (1974).

[56] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput. 39*, 9 (1990), 1175–1185.

[57] SHAPIRO, J. *EROS: A capability system*. PhD thesis, University of Pennsylvania, 1999.

[58] SHAPIRO, J. S., FARBER, D. J., AND SMITH, J. M. The measured performance of a fast local IPC. In *IWOOOS '96: Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)* (Washington, DC, USA, 1996), IEEE Computer Society, p. 89.

[59] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. Eros: a fast capability system. In *ACM Symposium on Operating Systems Principles (SOSP'99)* (December 1999), vol. 34, pp. 170–185.

[60] SINHA, A., SARAT, S., AND SHAPIRO, J. S. Network subsystems reloaded: a high-performance, defensible network subsystem. In *Proceedings of the USENIX Annual Technical Conference 2004* (2004), USENIX Association, pp. 19–19.

[61] STEINBERG, U., AND KAUER, B. Towards a scalable multiprocessor user-level environment. In *IIDS '10: Proceedings of the First Workshop on Isolation and Integration in Dependable Systems* (2010), ACM, pp. 1–6.

[62] STEINBERG, U., WOLTER, J., AND HÄRTIG, H. Fast component interaction for real-time systems. In *Proceedings of ECRTS'05* (July 2005), pp. 89–97.

[63] WULF, W. A., COHEN, E. S., CORWIN, W. M., JONES, A. K., LEVIN, R., PIERSON, C., AND POLLACK, F. J. Hydra: The kernel of a multiprocessor operating system. *Commun. ACM 17*, 6 (1974), 337–345.

Notes

[1]Resource allocation policies define how and when resources are divided between the tasks

[2]Named after the Greek philosopher Anaxagoras, who said: "Nothing is born or perishes, but already existing things combine, then separate anew", which can be seen as a summary of the resource security principles

[3](This "typecall" mechanism, invented in Hydra [42], proved to be the only one needed in many following capability systems [42, 59, 35]).

[4]Actually, when the frame types correspond to kernel objects (e.g. thread and domain), privileged operations on these objects is done using a capability that directly points to the object. This allows using the object without owning its memory frame, and direct access to the thread/domain kernel services. These services handle capability creation, copy, scheduling, inter-domain inter-processor interrupts...

[5]Furthermore, hard-coded timing constants are always a problem: what should be done if the extra time is too small?

[6]The difference is that it is easier to programs to know that they are in a recovery process

[7]Sometimes there is only one resource of a kind, for instance a keyboard